1. 代理模式

1.1 为什么要用代理模式

通常在编写一段代码的时候，我们可能会需要做一些与功能不是很相关的处理，如日志的记录，安全和事务的处理，这不是我们设计方法的主要目的，这是对面向对象的一种破坏。

这些辅助的代码会依赖于其他类，加深类之间的耦合，造成主体代码的可移植性降低

1.2 代理模式的作用

作为对象和使用者之间的中介，提供一种代理来实现对对象的访问。

代理模式一般设计三个角色：

抽象角色：作为真实对象和代理对象的接口，一般是抽象类或者接口，实现代理对象对实现了抽象类或者接口的一系列的统一代理

代理角色：代理对象内部含有真实对象的应用（多态实现），能通过内部引用代理真实对象的相同接口，并加入一些辅助功能。

真实角色：被代理的真实对象，在这里我们仅需要做我们必须做的事，一些相同的辅助功能交给代理对象去实现。

1.3 代理模式 demo

```
//抽象接口,实现了该接口的方法都可以被代理
public interface abstractObject {
    abstract public void doSomething();
}
```

```java
//真实被代理对象，做主体功能，准备工作交给代理对象做
public class RealObject implements abstractObject{

    @Override
    public void doSomething() {
        System.out.println("我在做一些重要的事情");
    }

}
```

```java
//代理对象，负责具体细节处理，实现主体代码与辅助代码的分离，降低耦合度，提高主体代码的复用性
public class ProxyObject {
    //将抽象角色的引用作为类属性用来持有一个被代理对象
    private abstractObject abstractObject;
    //通过构造方法，持有一个具体可以被代理对象
    public ProxyObject(com.zhangrui.basictest.abstractObject abstractObject) {
        this.abstractObject = abstractObject;
    }

    public void doSomething(){
        preDoSomenthing();
        abstractObject.doSomething();//被代理对象要做具体功能
        afterDoSomething();
    }

    public void preDoSomenthing(){
        System.out.println("准备中。。。。");
        //主体功能进行之前的预处理(如开启事务)
    }

    public void afterDoSomething(){
        System.out.println("收尾中。。。。");
        //做一些收尾工作(如提交事务)
    }

}
```

```java
public static void main(String[] args) {
    ProxyObject myProxy = new ProxyObject(new RealObject());
    myProxy.doSomething();
}
```

```
准备中。。。。
我在做一些重要的事情
收尾中。。。。
```

1.4 java 动态代理

java 动态代理只能代理接口实现，因为产生的代理类本身继承自 Proxy，受限于 java 的单继承

1.4.1 java.lang.reflect 介绍

InvocationHandler：

代理实例调用处理程序的接口

invoke

Object invoke(Object proxy,
Method method,
Object[] args)
throws Throwable

在代理实例上处理方法调用并返回结果。在与方法关联的代理实例上调用方法时，将在调用处理程序上调用此方法。

参数：

proxy - 在其上调用方法的代理实例

method - 对应于在代理实例上调用的接口方法的 Method 实例。Method 对象的声明类将是在其中声明方法的接口，该接口可以是代理类赖以继承方法的代理接口的超接口。

args - 包含传入代理实例上方法调用的参数值的对象数组，如果接口方法不使用参数，则为 null。基本类型的参数被包装在适当基本包装器类（如 java.lang.Integer 或 java.lang.Boolean）的实例中。

返回：

从代理实例的方法调用返回的值。如果接口方法的声明返回类型是基本类型，则此方法返回的值一定是相应基本包装对象类的实例；否则，它一定是可分配到声明返回类型的类型。如果此方法返回的值为 null 并且接口方法的返回类型是基本类型，则代理实例上的方法调用将抛出 NullPointerException。否则，如果此方法返回的值与上述接口方法的声明返回类型不兼容，则代理实例上的方法调用将抛出 ClassCastException。

抛出：

Throwable - 从代理实例上的方法调用抛出的异常。该异常的类型必须可以分配到在接口方法的 throws 子句中声明的任一异常类型或未经检查的异常类型 java.lang.RuntimeException 或 java.lang.Error。如果此方法抛出经过检查的异常，该异常不可分配到在接口方法的 throws 子句中声明的任一异常类型，代理实例的方法调用将抛出包含此方法曾抛出的异常的 UndeclaredThrowableException。

另请参见：

UndeclaredThrowableException

Proxy:

Proxy 提供用于创建动态代理类和实例的静态方法，它还是由这些方法创建的所有动态代理类的超类。

创建某一接口 Foo 的代理：

```
InvocationHandler handler = new MyInvocationHandler(...);
Class proxyClass = Proxy.getProxyClass(
    Foo.class.getClassLoader(), new Class[] { Foo.class });
Foo f = (Foo) proxyClass.
    getConstructor(new Class[] { InvocationHandler.class }).
    newInstance(new Object[] { handler });
```

或使用以下更简单的方法：

```
Foo f = (Foo) Proxy.newProxyInstance(Foo.class.getClassLoader(),
                                     new Class[] { Foo.class },
                                     handler);
```

构造方法摘要

protected	**Proxy**(InvocationHandler h) 使用其调用处理程序的指定值从子类（通常为动态代理类）构建新的 Proxy 实例。

方法摘要

static InvocationHandler	**getInvocationHandler**(Object proxy) 返回指定代理实例的调用处理程序。
static Class<?>	**getProxyClass**(ClassLoader loader, Class<?>... interfaces) 返回代理类的 java.lang.Class 对象，并向其提供类加载器和接口数组。
static boolean	**isProxyClass**(Class<?> cl) 当且仅当指定的类通过 getProxyClass 方法或 newProxyInstance 方法动态生成为代理类时，返回 true
static Object	**newProxyInstance**(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h) 返回一个指定接口的代理类实例，该接口可以将方法调用指派到指定的调用处理程序。

1.4.2 demo

```
public class DynamicProxy implements InvocationHandler{
    private Object realObject;

    public DynamicProxy(Object realObject) {
        super();
        this.realObject = realObject;
    }

    /**
     * proxy:代理类实例
     * method:代理类代理方法
     * agrs 代理方法参数
     * 返回被代理方法返回值
     */
    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
        preDoSomenthing();
        Object obj = method.invoke(realObject, args);  //动态代理调用方法，参数1：调用方法的对象
        afterDoSomething();
        return obj;
    }
    public void preDoSomenthing(){
        System.out.println("准备中。。。。");
        //主体功能进行之前的预处理(如开启事务)
    }

    public void afterDoSomething(){
        System.out.println("收尾中。。。。");
        //做一些收尾工作(如提交事务)
    }

}
```

```
public static void main(String[] args) {
    RealObject realObject = new RealObject();
    Class realObjectClass = realObject.getClass();
    InvocationHandler dynamicProxy = (InvocationHandler) new DynamicProxy(realObject);
    AbstractObject abstractObject =
            (AbstractObject)
            Proxy.newProxyInstance(realObjectClass.getClassLoader(),
                    realObjectClass.getInterfaces(), dynamicProxy);
    abstractObject.doSomething();
}
```

通过这种方式，被代理的对象(RealObject)可以在运行时动态改变，需要控制的接口 (AbstractObject 接口)可以在运行时改变，控制的方式(DynamicProxy 类)也可以动态改变，从而实现了非常灵活的动态代理关系

1.4.3 AOP 实现拦截切入

AOP（Aspect Oriented Programming）面向切面编程就是利用 java 动态代理实现的

两种方法：jdk 动态代理和 cglib 动态代理

名词介绍：

ASM:

java 字节码操纵框架，被用来动态生成类或者增强既有类的功能。ASM 可以直接产生二进制 class 文件，也可以在类被加载入 Java 虚拟机之前动态改变类行为。Java class 被存储在严格格式定义的 .class 文件里，这些类文件拥有足够的元数据来解析类中的所有元素：类名称、方法、属性以及 Java 字节码（指令）。ASM 从类文件中读入信息后，能够改变类行为，分析类信息，甚至能够根据用户要求生成新类。

CGLIB：

一个强大的，高性能，高质量的 Code 生成类库，它可以在运行期扩展 Java 类与实现 Java 接口。

JDK 的动态代理用起来非常简单，但它有一个限制，就是使用动态代理的对象必须实现一个或多个接口。如果想代理没有实现接口的继承的类，那么就可以使用 CGLIB 包

Cglib 是一个优秀的动态代理框架，它的底层使用 ASM 在内存中动

态的生成被代理类的子类，使用 CGLIB 即使代理类没有实现任何接口也可以实现动态代理功能。CGLIB 具有简单易用，它的运行速度要远远快于 JDK 的 Proxy 动态代理。

jdk 动态代理 1.4.2demo 已经演示过了

cglib 动态代理的实现：(导入 jar 包 cglib-nodep)

cglib 核心类：

（1）net.sf.cglib.core：

底层字节码处理类，他们大部分与 ASM 有关系。

（2）net.sf.cglib.transform：编译期或运行期类和类文件的转换。

（3）net.sf.cglib.proxy ：实现创建代理和方法拦截器的类。

（4）net.sf.cglib.reflect ：实现快速反射和 C#风格代理的类。

（5）net.sf.cglib.util：集合排序工具类。

（6）net.sf.cglib.beans：JavaBean 相关的工具类。

net.sf.cglib.proxy.MethodInterceptor 接口是最通用的回调（callback）类型，它经常被基于代理的 AOP 用来实现拦截（intercept）方法的调用。这个接口只定义了一个方法

```
 public Object intercept(
Object object,
 java.lang.reflect.Method method,
 Object[] args,
MethodProxy proxy) throws Throwable;
```

第一个参数是被代理对象，第二个是代理的方法，第三个是方法参数

demo:

```
public class CglibProxy implements MethodInterceptor{

    @Override
    public Object intercept(Object arg0, Method arg1, Object[] arg2,
            MethodProxy arg3) throws Throwable {
        //用arg3而不是arg1，可以提高性能
        preDoSomenthing();
        Object result = arg3.invokeSuper(arg0, arg2);
        afterDoSomething();
        return result;

    }

    public void preDoSomenthing(){
        System.out.println("准备中。。。。");
        //主体功能进行之前的预处理(如开启事务)
    }

    public void afterDoSomething(){
        System.out.println("收尾中。。。。");
        //做一些收尾工作(如提交事务)
    }
}
```

```
public class Test1 {
    public static void main(String[] args) {
        CglibProxy cglibProxy = new CglibProxy();
        Enhancer enhancer = new Enhancer();
        enhancer.setSuperclass(RealObject.class);
        enhancer.setCallback(cglibProxy);
        AbstractObject abstractObject = (AbstractObject)enhancer.create();
        abstractObject.doSomething();
    }
}
```

1.5 cglib 动态代理详解

http://blog.csdn.net/zhoudaxia/article/details/30591941

创建一个具体类的代理时，通常要用到的 CGLIB 包的 APIs：

net.sf.cglib.proxy.Callback 接口：在 CGLIB 包中是一个很关键的接口，所有被 net.sf.cglib.proxy.Enhancer 类调用的回调（callback）接口都要继承这个接口。

net.sf.cglib.proxy.MethodInterceptor 接口：是最通用的回调（callback）类型，它经常被 AOP 用来实现拦截（intercept）方法的调用。这个接口只定义了一个方法。

```
public Object intercept(Object object, java.lang.reflect.Method method, Object[] args, MethodProxy proxy) throws Throwable;
```

net.sf.cglib.proxy.MethodInterceptor 能够满足任何的拦截（interception）需要，当对有些情况下可能过度。为了简化和提高性能，CGLIB 包提供了一些专门的回调（callback）类型。例如：

net.sf.cglib.proxy.FixedValue：为提高性能，FixedValue 回调对强制某一特别方法返回固定值是有用的。

net.sf.cglib.proxy.NoOp：NoOp 回调把对方法调用直接委派到这个方法在父类中的实现。

net.sf.cglib.proxy.LazyLoader：当实际的对象需要延迟装载时，可以使用 LazyLoader 回调。一旦实际对象被装载，它将被每一个调用代理对象的方法使用。

net.sf.cglib.proxy.Dispatcher：Dispathcer 回调和 LazyLoader 回调有相同的特点，不同的是，当代理方法被调用时，装载对象的方法也总要被调用。

net.sf.cglib.proxy.ProxyRefDispatcher：ProxyRefDispatcher 回调和 Dispatcher 一样，不同的是，它可以把代理对象作为装载对象方法的一个参数传递。

代理类的所以方法经常会用到回调（callback），当然你也可以使用 net.sf.cglib.proxy.CallbackFilter 有选择的对一些方法使用回调

（callback），这种考虑周详的控制特性在 JDK 的动态代理中是没有的。在 JDK 代理中，对 java.lang.reflect.InvocationHandler 方法的调用对代理类的所有方法都有效。

CGLIB 的代理包也对 net.sf.cglib.proxy.Mixin 提供支持。基本上，它允许多个对象被绑定到一个单一的大对象。在代理中对方法的调用委托到下面相应的对象中。

2. 策略设计模式

方法操作接口而不是类，可以实现类与类之间的完全解耦，创建一个能够根据所传递的参数对象不同而具有不同行为的方法，被称为策略设计模式。方法包含所要执行的算法中固定不变的部分，而策略包含变化的部分。策略就是传递进去的对象，它包含要执行的代码。方法对不同策略进行封装，提供统一使用接口。

2.1 策略模式的结构

•封装类：也叫上下文，对策略进行二次封装，目的是避免高层模块对策略的直接调用。

•抽象策略：通常情况下为一个接口，当各个实现类中存在着重复的逻辑时，则使用抽象类来封装这部分公共的代码，此时，策略模式看上去更像是模版方法模式。

•具体策略：具体策略角色通常由一组封装了算法的类来担任，这些类之间可以根据需要自由替换。

2.2 demo

将变化的部分抽象出来形成接口，每个接口可以有许多具体的实现。不同鸟儿不同的具体行为该如何实现不是鸟儿本身做的，而是它与生俱来就有的，接口的棘突实现作为他的属性，是通过组合的方式实现的，实现了代码的复用。鸟的共性可以做成一个抽象类，让所有鸟儿继承，以实现多态。我们给出同一个接口，你给我一个鸟儿，我就告诉你鸟的颜色，鸟怎么飞，这是一个统一的接口，我并不关心其本省是怎么实现的，具体的实现细节被封装起来了，我提供的只是一个对策略的二次封装。

3. 适配器设计模式

有时候一个不是你自己创建的，而是你通过导入库得到的。此时你要想实现类与类之间的解耦合。你无法通过是这个类来实现某一接口，通过多态来实现代码复用，那么此时就可以选择采用适配器模式。将一个类的接口转换成客户希望的另外一个接口。Adapter 模式使得原本由于接口不兼容而不能一起工作的那些类可以在一起工作。

3.1 模式中的角色

目标接口（Target）：客户所期待的接口。目标可以是具体的或抽象的类，也可以是接口。

需要适配的类（Adaptee）：需要适配的类或适配者类。

适配器（Adapter）：通过包装一个需要适配的对象，把原接口转换成

目标接口。

3.2 实现方式

适配器对象中内部包装一个需要适配的类，适配器实现目标接口。通过操作适配间接操作待适配的类。

3.3 demo

```java
//这是目标接口
public interface Target {
    public void doSomething();
}
```

```java
//这是待适配的类，由于种种原因（这个类不是你创建的，是你导入的）不能实现target接口。

public class Client {
    public void doSomething(){
        System.out.println("我要做一些事情");
    }
}
```

```java
public class ClientAdapter implements Target{
    private Client client;

    public ClientAdapter(Client client) {
        super();
        this.client = client;
    }

    @Override
    public void doSomething() {
        client.doSomething();
    }

    public static void main(String[] args) {
        //通过这种方式就能实现client和其他Target接口的实现一样调用统一的doSomenThing方法，实现代码的复用
        ClientAdapter adapter = new ClientAdapter(new Client());
        adapter.doSomething();
    }

}
```

4.工厂模式

5. 装饰器设计模式